Coordination of a Parallel Proposition Solver

C.T.H. Everaars and B. Lisser

Software Engineering (SEN)

SEN-R9832 December 1998

Report SEN-R9832
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Coordination of a Parallel Proposition Solver

C.T.H. Everaars and B. Lisser
Kees.Everaars@cwi.nl and Bert.Lisser@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

In this paper we describe an experiment in which **MANIFOLD** is used to coordinate the interprocess communication in a parallelized proposition solver. **MANIFOLD** is very well suited for applications involving dynamic process creation and dynamically changing (ir)regular communication patterns among sets of independent concurrent cooperating processes. The idea in this case study is simple. The proposition solver consists of a fixed numbers of separate processing units which communicate with each other such that the output of one serves as the input for the other. Because one of the processing units performs a computation intensive job, we introduce a master/worker protocol to divide its computations. We show that this protocol implemented in **MANIFOLD** adds another hierarchic layer to the application but leaves the previous layers intact. This modularity of **MANIFOLD** offers the possibility to introduce concurrency step by step. We also verify the implementation of the proposition solver using a simple family of assertions and give some performance results.

1991 Computing Reviews Classification System: D.3.3, D.1.3, D.3.2, F.1.2, D.2.1.
1991 Mathematics Subject Classification System: 68N15, 68Q10.
Keywords and Phrases: Parallel Computing, Distributed Computing, Coordination Languages, Proposition Solvers.
Note: joint work between SEN2 "Specification and Analysis of Embedded Systems" and SEN3 " Coordination Languages"

1. Introduction

Correctness assertions, about e.g., the safety of railway platforms, the working of embedded systems, or computer programs are frequently formulated as statements in propositional logic [GKvV95]. Such a proposition, formulated in some specification language, can then form the input for a proposition solver that tests the correctness of the assertion. In general, we can see a proposition solver as a system that solves equations such as $p(\vec{x}) = true$, where $\vec{x}$ stands for the free boolean variables in proposition p. The satisfiability of the proposition p is checked and a solution of the equation is computed, provided that p is satisfiable. In [LW98] we can find a description of a library containing formal specifications of components that can be used as building blocks for a generic solver tool for logical propositions.

In this paper we describe how the actual implementation of the solver components are used in cooperation with each other to build a real working parallel proposition solver. The different components of the solver were implemented (in C) by a group of researchers in the department of Software Engineering at CWI (Centrum voor Wiskunde en Informatica) in the Netherlands in cooperation with the Dutch Railways for checking railway safety systems [GKvV95]. Finishing the implementation (and debugging) of the individual components, they looked for an efficient interprocess communication tool that could take care for the necessary communication channels between the different components. There are many different languages and programming tools available that can be used to implement this kind of communication, each representing a different approach to communication. Normally, languages like Compositional C++, High Performance Fortran, Fortran M, Concurrent C(++) or tools like MPI, PVM, and PARMACS are used (see [Arb97] for some critical notes on these languages and tools). There is, however, a promising novel approach: the application of *coordination* languages [GC92].

In this paper we describe how the coordination language MANIFOLD is used for the interprocess communication between the different components of this proposition solver. MANIFOLD is a coordination language developed at the CWI in the Netherlands. It is very well suited for applications involving dynamic process creation and dynamically changing (ir)regular communication patterns among sets of independent concurrent cooperating processes [Arb95, Arb96].

The rest of this paper is organized as follows. In section 2 we give a brief introduction to the MANIFOLD language by discussing a "toy" application. Apart from showing some of the syntax and semantics of MANIFOLD we find in this example already, in nutshell, most elements of the MANIFOLD system[1]. In section 3 we give a formal description of the proposition solver and its different functional units, and in section 4 we describe its actual implementation. Because one of the functional units is responsible for the heavy computational work in the proposition solver, we apply a coarse-grain restructuring on that functional unit. This adds another hierarchic layer to the application which we describe in section 5. In section 6, we test with a family of assertions the proposition solver and give some performance results. Finally, the conclusion of the paper is in section 7.

2. The Manifold Coordination Language

In this section, we briefly introduce MANIFOLD. Programming in MANIFOLD is a game of dynamically creating process instances and (re)connecting the ports of some processes via streams (asynchronous channels), in reaction to observed event occurrences. This style reflects the way one programmer might discuss his interprocess communication application to another programmer by telephone (let process a connect process b with process c so that c can get its input; when process b receives event e, broadcast by process c, react on that by doing this and that; etc.). As is already clear from this phone call, processes in MANIFOLD do not explicitly send a message to or receive a message from another process. Processes in MANIFOLD are treated as black-box workers that can only read or write through the openings (called ports) in their own bounding walls. It is always a third party - a coordinator process that is called a manager - that is responsible for setting up the communication channel (in MANIFOLD called a stream) between the output port of one process and the input port of another process, so that data can flow through it. This setting up of the communication from the *outside* is very typical for MANIFOLD and has several advantages. An important advantage is that it results in a clear separation between the modules responsible for computation and the modules responsible for coordination, and therefore also strengthens the modularity and enhances the re-usability of both types of modules (see [Arb96], [Arb97]).

We now illustrate MANIFOLD through an example that implements a game of table tennis between "you " and "me". We give the MANIFOLD source file of this example below (lines numbers have been added).

```
 1 #include "MBL.h"
 2
 3 #include "rdid.h"
 4
 5 event ping, pong.
 6
 7 /*************************************************/
 8 manifold ping_pong_player(event e)
 9 {
10   auto process v is variable.
11
12   begin: v = input; MES(v); raise(e); output = v; post(begin).
13 }
14
15 /*************************************************/
16 manifold umpire
17 {
18   process you is ping_pong_player(ping).
19   process me  is ping_pong_player(pong).
20
21   begin: (MES("Lets start ping-ponging"),
22           activate(you, me), "ball" -> you, terminated(void)).
23
24   ping: you -> me.
25
26   pong: me -> you.
27 }
```

[1]For more information, refer to our html pages located at http://www.cwi.nl/cwi/projects/manifold.html.

```
28
29 /************************************************/
30 manifold Main
31 {
32   begin: umpire.
33 }
```

This code describes three manifolds (i.e., process types) named `ping_pong_player`, `umpire` and `Main` (respectively lines 8-13, 16-27, 30-33). A manifold is a template from which we can make process instances. A process instance always has an event memory in which itself or other process instances can put event occurrences. MANIFOLD is an event driven language which means that once a process instance detects an event occurrence in its event memory, the process instance makes a transition out of its current state to the state that is labeled with the name of that event occurrence. Switching to a state also means that the streams that were constructed in the former state are broken down (see [Arb96] for the details). In MANIFOLD syntax, a state looks like "`event_name: actions to be executed.`" and its semantic is "switch to this state when there is an event with this name in the event memory and execute the actions". The most important (primitive) actions are (1) creating and activating process instances, (2) broadcasting events (with the action `raise`) or putting it in a process' own event memory (with the action `post`), (3) connecting processes to each other by setting up streams between their ports (by the action denoted by the arrow `->`).

A manifold written in the MANIFOLD languages (we can also write them in a traditional programming language such as C) always has the following structure. After the word `manifold` a name is given (line 8: `ping_pong_player`) followed by some optional parameters (line 8: `event e`). After this first line, comes the body of the manifold (lines 9-13). The body of a manifold is a block enclosed in a pair of braces (for the ping-pong player, lines 9 and 13) and contains some optional global declarations (line 10 for the ping-pong player, lines 18 and 19 for the umpire) followed by one or more states to which an instance of the manifold can jump when there is a suitable event in its event memory. In our case, we see that the `ping_pong_player` and `Main` manifolds (and thus in the instances) have only the `begin` state. The umpire, on the other hand, has three different states: the `begin`, `ping`, and `pong` states. A `begin` state in MANIFOLD is something special. Activation of a process instance automatically puts an occurrence of the special high priority `begin` event in the event memory of that process, which results in an initial state transition to the begin state.

In our artificial game of table tennis, we consider a ping-pong player as someone who sequentially performs in his begin state the following actions (sequential execution is syntactically denoted by the connective ";" between the actions):

- It reads (catches) a ball (line 12: `v=input`) from its input port and stores it in a variable (line 10).
- It shows the ball on the screen (line 12: MES(v)).
- It broadcasts the event it receives as a parameter (line 8) to signal the umpire that it is intends to write (return) the ball to its output port (line 12: `raise(e)`).
- It puts the `begin` event in its own event memory (line 12: `post(begin)`) which results in another round of execution of the actions in the `begin` state.

The umpire manifold can be described as follows:

- Using in the global declaration part, the syntactic construct "`process x is y.`", it creates two processes named "you" and "me" as instances of the ping-pong player manifold (line 18, 19). Note that the actual parameters in the process creation "you" and "me" are respectively `ping` and `pong` so that "you" always raises `ping` and "me" always raises `pong` (line 12: `raise(e)`).
- In the `begin` state, the umpire shows the message "`Lets start ping-pongging`" on the screen (line 21), activates the two process instances "you" and "me" (line 22), gives a ball

to "you" (line 22: `"ball" -> you`) and waits (denoted by `terminated(void)` on line 22) until it detects one of the global events (declared on line 5).

- Because "you" starts returning the ball and raises (broadcasts) the `ping` event (line 12), the first event found in the event memory of the umpire is `ping` which causes a state transition from the `begin` state to the `ping` state. In this state a stream is created between "you" and "me" so that the ball can flow through this stream to "me" (line 24: `you -> me`).

- The process instance "me" behaves the same way as "you" except that it raises the event `pong` to signal the umpire that the ball is written to its output port. In reaction to this event, the umpire makes a state transition out of the `ping` state, which results in breaking the connection between "you" and "me", and enters the `pong` state where a new connection between "me" and "you" (line 26: `me -> you`) is created through which the ball can flow back to "you". It is clear that this table tennis game never stops.

The third manifold is `Main`. The name `Main` is indeed special in MANIFOLD: there must be a manifold with that name in every MANIFOLD application and an automatically created instance of this manifold called `main` is the first process that is started up in an application. In our case the `Main` manifold has only the `begin` state in which it automatically creates an process instance of the umpire manifold, just by calling the umpire by name (line 32). This implicit process instantiation is an alternative to the explicit creation of a process (as we did on line 18) and explicit activation (as we did on line 22).

Our table tennis program is very artificial and one might think of many other ways to implement it. For instance we can set up the stream connections between the ping-pong players in the umpire in just one state, in which case there is no need to do state transitions and thus there is no need to break the stream connections between the ping-pong players again and again. We chose to implement it as we did because it clearly shows the dynamic changing of connections on the beat of event occurrences which is an important aspect of MANIFOLD's event driven nature.

Process instances in a MANIFOLD application always run as separate threads (light weight processes [NBF96]) within an operating-system level process. This latter process (heavy weight) is called a task instance in MANIFOLD. The way process instances are bundled in task instances influences the mechanism that is actually used for the data transport in streams (the `->`). The bundling can be done automatically as well as controlled. When we take care that all the process instances of the MANIFOLD application run as threads in the same task instance, the effective data transport in the streams between these process instances is implemented in shared memory. In that case we in fact play "shared memory table tennis". We can also bundle the process instances in such a way that each ping-pong player and the umpire is housed in a separate task instance. In that case the effective data transport in streams between the process instances is implemented using Unix sockets. The mapping of process instances in task instances is considered to be a separate stage in the application construction. This mapping is described in a file which is the input for the MANIFOLD linker MLINK. An example of such an the input file is shown below (line numbers have been added).

```
1 {task *
2  {load 10}
3  {weight ping_pong_player 10}
4  {weight umpire 10}
5 }
6
7 {task pingpong
8   {include pingpong.o}
9 }
```

In this file we specify that a task instance is considered to be "full" when its load exceeds 10 (line 2) and that the weight of an instance of the ping-pong player or umpire is also 10 (line 3, 4). The net result of this is that each task instance will house only one thread and thus instances of ping-pong player and the umpire end up in different instances of the task named `pingpong` (line 7). The primary output of the manifold linker is a *makefile*, plus a number of C source files necessary to provide the inter-task information.

This *makefile* is meant to be used as a black-box by recursive make commands in programmer-defined makefiles that finally create the executable files suitable for the appropriate platforms. With this the task composition stage comes to an end and the final stage in application construction can start. This is the run-time configuration stage, in which we define the mapping of tasks to hosts. This mapping too is described in a file which is the input for the MANIFOLD run-time configurator CONFIG. An example of such an the input file is shown below.

```
{host host1 sampan.cwi.nl}
{host host2 pont.cwi.nl}
{host host3 opduwer.cwi.nl}
{locus pingpong $host1 $host2 $host3}
```

Here we define three variables `host1`, `host2` and `host3`, which we set to respectively `sampan.cwi.nl`, `pont.cwi.nl` and `opduwer.cwi.nl`. These are the names of computers located at different places and connected via a network. The last line in the file states that task instances (in our case three) of the pingpong task can be started on any of these three machines.

Note that the different mappings in the task composition stage and the run-time configuration stage do not affect the semantics of the MANIFOLD source code.

Running the ping-pong program using the task composition and run-time configuration described above forms a "distributed table tennis game" and gives the following output.

```
262159 112 pingpong umpire() pingpong.m 21 -> Lets start ping-ponging
786433 63 pingpong ping_pong_player(event) pingpong.m 12 -> ball
524289 63 pingpong ping_pong_player(event) pingpong.m 12 -> ball
786433 63 pingpong ping_pong_player(event) pingpong.m 12 -> ball
524289 63 pingpong ping_pong_player(event) pingpong.m 12 -> ball
786433 63 pingpong ping_pong_player(event) pingpong.m 12 -> ball
Etc.
```

Each of these output lines has the following structure. It starts with a long label followed by a `->` before the actual message (the value of the argument of `MES`). The label shows respectively the identification of the task instance, the identification of the process instance, the name of the task, the name of the manifold, and the name of the MANIFOLD source file and the line number where `MES` is called. With such a label in front of the actual message, we always know *who* is printing *what* and *where*.

The MANIFOLD system runs on multiple platforms and consists of a compiler (MC), a linker (MLINK), a run-time configurator (CONFIG), a run-time system library, a number of utility programs, and libraries of built-in and predefined processes of general interest. Presently, it runs on IBM RS60000 AIX, IBM SP1/2, Solaris, Linux, Cray, and SGI IRIX.

3. A Formal Description of the Proposition Solver

In this section we describe the solver components we use to build a parallel proposition solver that solves closed quantified propositions. For a full description of the components we refer to the formal specifications in [LW98] and for an elementary introduction literature on logic we refer to [NS93]. The task of solving closed quantified propositions can be divided in four transformations which will be done sequentially in such a way that the output of one transformation serves as input for the next transformation.

The sequence of transformations a proposition has to go through, is shown in figure 1 and is as follows (for details we again refer to [LW98]).

- **PRENEX** [2] is the component that transforms a proposition to a prenex normal form (i.e., a quantifier free proposition preceded by a row of quantifiers).
- **CNF** (Conjunctive Normal Form) is the component that transforms a proposition in the prenex normal form to a quantified conjunctive normal form (i.e., a conjunctive normal form preceded by a row of quantifiers). This is done in the following way. First the proposition in the prenex normal form is split by a separate splitting component named **SPLIT** in a

[2] From here on in this paper we use small letters for processes and software components and capital letters for transformations and manifolds.

quantifier free part and a list of quantifiers. The quantifier free part is transformed by a separate component named **CNFHH** (Conjunctive Normal Form HeerHugo) to an existential conjunctive normal form (i.e., a conjunctive normal form preceded by a row of existential quantifiers). This output and the list of quantifiers are joined together by a component named **JOIN**, which results in a quantified conjunctive normal form.

- **UQE** (Universal Quantifier Eliminator) is the component that transforms quantified conjunctive normal forms to existential conjunctive normal forms. This transformation is done by the rule

$$\forall x.\exists \vec{y}.\phi(x,\vec{y}) = \exists \vec{y}_1.\phi(\mathit{false},\vec{y}_1) \wedge \exists \vec{y}_2.\phi(\mathit{true},\vec{y}_2) = \exists \vec{y}_1,\vec{y}_2.(\phi(\mathit{false},\vec{y}_1) \wedge \phi(\mathit{true},\vec{y}_2))$$

  and is the most computation intensive part of the proposition solver.

- **HH** (HeerHugo) is the component that simplifies existential conjunctive normal forms. Heerhugo has been developed at the University of Utrecht and CWI for checking railway safety systems [Gro97].

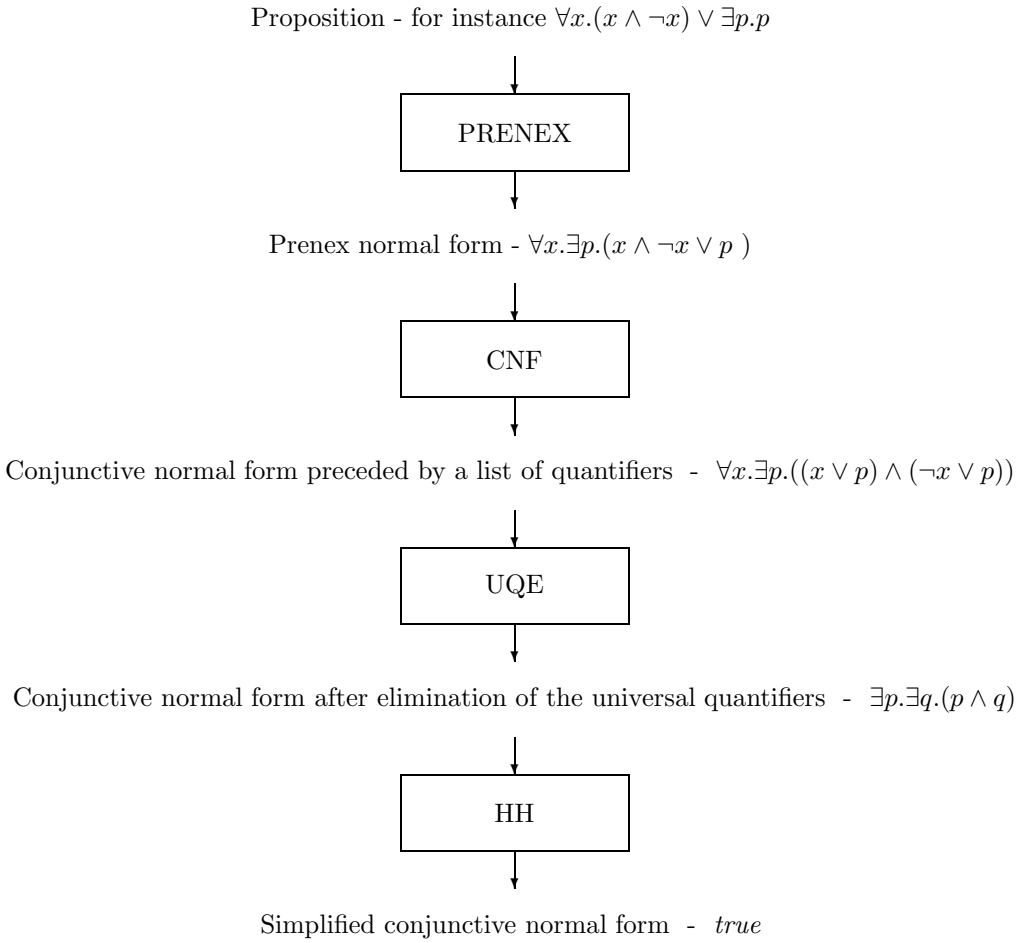

Figure 1: Transformations done at proposition solving

4. The Coordination in the Proposition Solver

The communication patterns in the proposition solver as given in section 3 in fact describe a pipeline model. It is a sequence of functional units ("stages") which performs a task in several steps. Each functional unit takes its input and produces output, where the output of one serves as the input for next stage. This arrangement allows all the stages to work in parallel, thus giving greater throughput than if each input had to pass through the whole pipeline before the next input could enter. We can visualize such a system as a set of nodes that are connected by streams in which data flows in one direction.

Each functional unit as described in section 3 has been implemented as a separate software component written in C. The generic data type used in the implementation to store a proposition, is a tree structure and is called a term. In the software components all kinds of operations are done on these terms, e.g., composing and decomposing terms, reading and writing terms, etc. For this we use the primitives defined in the term library of ToolBus [BK96].

Because the terms we communicate among the software components are very big (sometimes more than 7 megabytes) it is obvious that we want the interprocess communication in shared memory. This way it is sufficient to transport the pointer to a term from one software component to another. As already stated, transporting data in MANIFOLD is special. In MANIFOLD processes do not read and write *directly* from and to other processes, but they read and write from and to their own ports. It is a third party (a coordinator) that sets up, from the *outside*, the connections between them. For the manifolds written in C (also called atomic processes) we do this reading and writing to and from ports using, respectively, the functions `AP_PortRemoveUnit` and `AP_PortPlaceUnit` from the atomic process interface library. This is a standard MANIFOLD library with many C functions, which allows access to the MANIFOLD world. Besides the standard ports named `input`, `output`, and `error` we also have ports with user defined names, as we will see.

In the actual implementation of the proposition solver we need two additional software components; one for reading the assertion we want to verify (i.e., a term) and another for writing the true value of the assertion to the screen. These are named, respectively, **rd** and **pr** (Print Result). Below, we enumerate the different software components, describe them using MANIFOLD terminology, and show the stream connections between them in figure 2.

- **rd** reads a term representing a quantified closed proposition from a file and writes a pointer to this term to its own output port.
- **prenex** reads a term pointer to a quantified closed proposition from its input port, transforms the term to its prenex normal form, and writes a pointer to this result to its own output port.
- **cnf** reads a pointer to a term in prenex normal form, transforms the term to its quantified conjunctive normal form, and writes a pointer to this result to its own output port. **cnf** does not do the work by itself but behaves as a manager that delegates the work to others. The workers coordinated by **cnf** are:
  - **split** reads from its input port a pointer to a term in prenex normal form (this is the same pointer read by **cnf** from its input port), splits the term into a quantifier-free part and a list of quantifiers, the pointers to which are written to two user-defined output port `qfprop` (Quantifier-Free PROPosition) and `qvarlist` (Quantifier VARiable LIST), respectively.
  - **cnfhh** reads a pointer to a quantifier-free proposition, transforms the term to an existential conjunctive normal form, and writes a pointer to this result to its own output port.
  - **join** reads from a user defined input port named `qvarlist`, a pointer to a term representing a list of free quantifiers, reads from another user defined input port named `ecnf1` (Existential Conjunctive Normal Form 1) a pointer to a term representing an

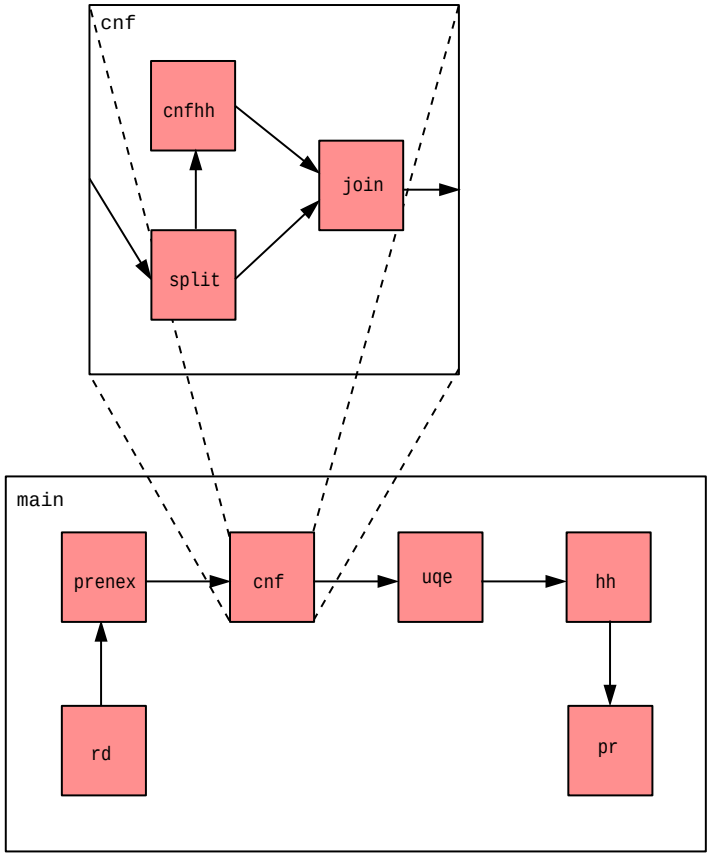

Figure 2: The network topology of the proposition solver

existential conjunctive normal form, joins these terms together to obtain a term representing a quantified conjunctive normal form, and writes a pointer to this result to its own output port.

- **uqe** reads a pointer to a term in quantified conjunctive normal form from its input port, transforms it to its existential conjunctive normal form, and writes a pointer to this result to it to its own output port.
- **hh** reads a pointer to a term in existential conjunctive normal form from its input port, transforms it to a simpler existential conjunctive normal form, and writes a pointer to this result to its own output port.
- **pr** reads a pointer to a term in existential conjunctive normal form, evaluates the value of this term and prints its value, which can be "true", "false" or "I don't know" to the screen.

The MANIFOLD source code of the proposition solver is very simple and is shown below.

```
 1 //pragma include "hugo4.ato.h"
 2
 3 #include "MBL.h"
 4
 5 #include "rdid.h"
 6
 7 manner init_tb atomic.
 8
 9 manifold RD(port in filename) atomic {internal.}.
10
11 manifold PRENEX() atomic {internal.}.
12
13 manifold SPLIT()
14   port out qfprop.
15   port out qvarlist.
```

```
16   atomic {internal.}.
17
18 manifold CNFHH() atomic {internal.}.
19
20 manifold JOIN()
21   port in qvarlist.
22   port in ecnf1.
23   atomic {internal.}.
24
25 manifold UQE() atomic {internal.}.
26
27 manifold HH() atomic {internal.}.
28
29 manifold PR() atomic {internal.}.
30
31 /*************************************************/
32 manifold CNF()
33 {
34   process split is SPLIT.
35   process cnfhh is CNFHH.
36   process join is JOIN.
37   begin:
38     (
39       MES("begin"),
40       activate(split, cnfhh, join),
41       input -> split,
42                split.qfprop -> cnfhh -> join.ecnf1,
43                split.qvarlist -> join.qvarlist,
44       join -> output
45     ).
46
47   end: MES("end").
48
49 }
50 /*************************************************/
51 manifold Main(process arg)
52 {
53   process rd is RD(tuplepick(arg, 2)).
54   process prenex is PRENEX.
55   process cnf is CNF.
56   process uqe is UQE.
57   process hh is HH.
58   process pr is PR.
59
60   begin:
61     (
62       MES("begin"),
63       init_tb,
64       activate(rd, prenex, cnf, uqe, hh, pr),
65       rd -> prenex -> cnf -> uqe -> hh -> pr
66     ).
67
68   end: MES("end").
69 }
```

The source code is in principle no more than the declarations of the different software components with their ports and the specification of the connections among them at each of the hierarchic layers. We now walk through the code.

With the pragma on line 1 we can verify the prototyping of the manifolds written in C with the declarations given for them in manifold source files. A mismatch will result in a syntax error issued by the C compiler.

On lines 3 and 5 we include some .h files for predefined processes and subprograms used in this source code.

Line 7 is the declaration for a subprogram written in C (denoted by the keyword atomic). It is used for initializing the term library of ToolBus [BK96].

Hereafter we see the declarations of eight manifolds implemented as atomic processes written in C (lines 9, 11, 13, 18, 20, 25, 27, and 29) and two manifolds written in the manifold language (lines 32 and 51). The `Main` manifold (line 51) and the `RD` (line 9) are the only manifolds in the source file that have arguments. The argument of `Main` is used to pass the filename, containing the proposition(s) we want to verify, from the Unix command line into the MANIFOLD world. The way we do that in MANIFOLD is analogous to the way we do such things in ANSI C. The argument in the `RD` manifold is meant to pass a filename to the underlying C function so that it can open this file and read a proposition from it.

All the manifolds in this source code have only the standard set of ports (`input`, `output` and `error`) except the `SPLIT` and `JOIN` manifolds. These two, as explained earlier, have additional user-defined ports. `SPLIT` has two additional *output* ports named `qfprop` and `qvarlist` (lines 14 and 15) and `JOIN` has two additional *input* ports named `qvarlist` and `ecnf1` (lines 21 and 22).

The `CNF` manifold written in the MANIFOLD language is a manager that coordinates the three

worker manifolds `SPLIT`, `CNFHH`, and `JOIN`. The workers are created in the global declaration part of `CNF` (lines 34-36). In the `begin` state of `CNF` we print a message to the screen to indicate that we are in the `begin` state (line 39), we activate process instances (line 40), and set up the desired connections (lines 41-44) as shown in figure 2. In MANIFOLD we use the notation $p.i$ to refer to port i of the process instance p. Furthermore, $p \rightarrow q$ means the same as $p.output \rightarrow q.input$ and to refer to the standard ports of a process instance from inside the process itself, we use the words `input`, `output`, and `error`. Thus line 41 means: connect the standard input port of (in instance of) `CNF` to (the input port of) `split` and line 42 means connect the `qfprop` output port of `split` to (the standard input port of) `cnfhh` and (the standard output port of) `cnfhh` to the `ecnf1` input port of `join`.

We also have added an `end` state in the `CNF` manifold (line 47). We switch to this state when all the connections between the different process instances in the `begin` state are broken (each at least on one side).

Note that `CNF` is a real manager. He delegates his input to others (line 41: `input ->`) and presents their output as coming from himself (line 44: `-> output`).

In the `Main` manifold, we create in its declaration part the process instances we need and we also have a `begin` and an `end` state. The actual argument used in the creation of process `rd` (line 53) is a filename that contains the proposition to be verified by the proposition solver. This filename is picked out (with the predefined process `tuplepick`) as the second argument in a list of arguments that has been typed in on the Unix command line. This list of arguments is know in `Main` via the argument `arg` (line 51).

In the `end` state, `Main` prints a message (line 62), initializes the term library (line 63), activates the process instances (line 64) and sets up the connections among them as shown in figure 2.

A call to a makefile creates the executable for this MANIFOLD application which is named `hugo`.

When we type in `hugo x` on the command line, where `x` is a filename that contains a proposition then the proposition solver calculates its truth value.

This coordination protocol is all that is needed for this application and the application runs fine as long as we wait and feed it a new proposition only after the previous one has been handled. As soon as we allow more than one term in the pipeline, the proposition solver crashes with a fatal signal caused by a bad memory reference. Indeed, there is nothing wrong with our protocol, nor with the solver components, per se. The problem is in the term library used in this application. This term library is a collection of string manipulating functions that was originally developed as a component of ToolBus [BK96]. These functions (and ToolBus) were developed without any regards for threads, i.e., they were not implemented in a "thread friendly" style. Essentially, to be "thread friendly", functions must be reentrant and generally avoid references to global variables, except through fine-grain locks. Such functions can be used in multi-threaded applications, and especially on proper multi-processor platforms, this can significantly improve performance. There are two alternatives for rectifying this problem: (1) make the functions reentrant and introduce the necessary fine-grain locking to make the term library thread-safe; and (2) introduce a coarse-grain lock to regulate the access to the entire library. Both alternatives are semantically correct and avoid the crash. The first involves a good deal of rewriting of the library functions. The second is easier to implement, but because the library is heavily used in this application, it inhibits performance improvements (on multiprocessor platforms) by sequentializing term manipulation at the overly-coarse level of library access. Because using the ToolBus term library in this application suffers from some additional serious drawbacks (many redundant format conversions, inefficient memory utilization, slow garbage collection, and problems with handling big terms) it was decided to replace the library with a new one. However, we use the second alternative here as interim solution, and focus on other coordination issues in this application. This leads to next issue.

The **UQE** transformation performs a computation intensive simplification on a list of terms. In principle, we can cut this list into a number of pieces and perform the simplification transformation on each of the sub-lists. When we take care that the transformations on the sub-lists are carried out in separate worker processes, then they can run in parallel, as separate threads executing on different processors on multi-processor hardware. In the next section, we describe this restructuring of the **UQE** transformation using a master/worker protocol implemented in MANIFOLD.

5. Restructuring the **UQE** Transformation

The introduction of the master/worker protocol in the **UQE** transformation adds another hierarchic layer to the application but leaves, as we can see in figure 3, the previous layers intact. New

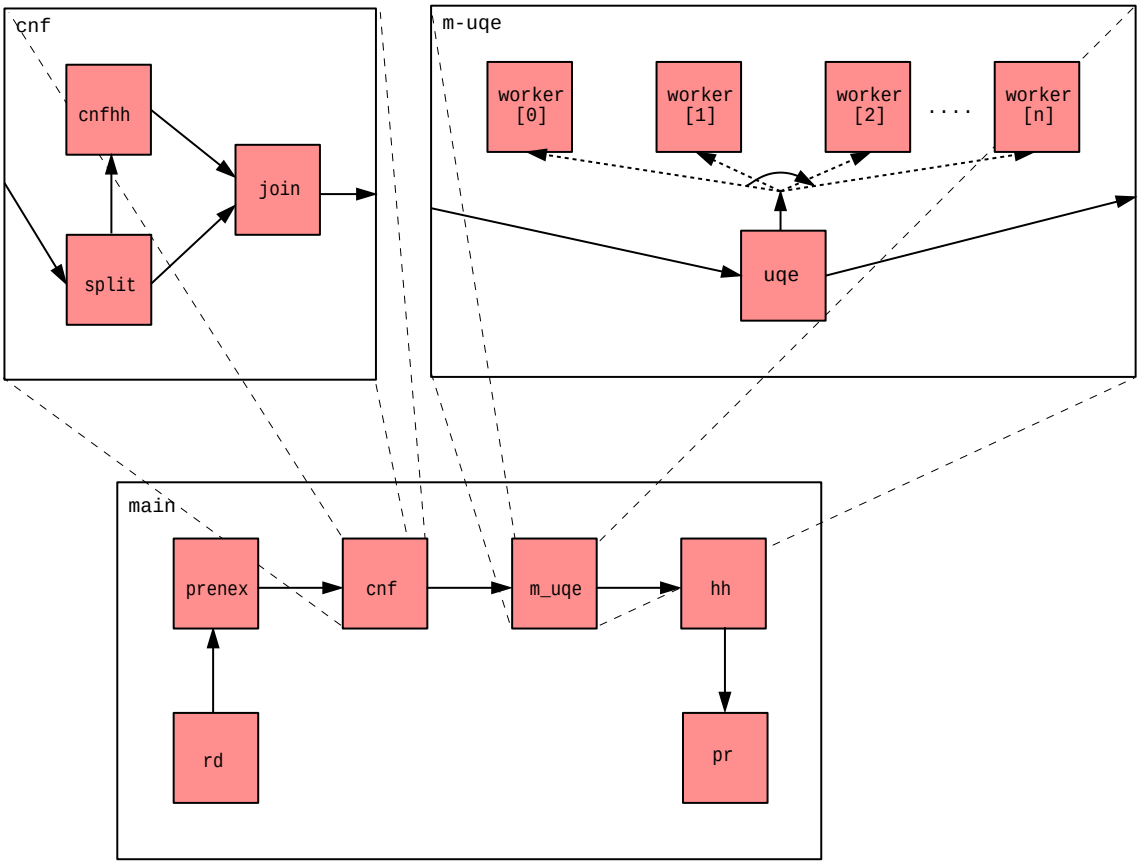

Figure 3: The new network topology of the proposition solver

in this figure are the manifold `M_UQE`, in which the master/worker protocol is implemented, and another manifold named `WORKER`, which performs a simplification transformation on a sub-list. We show the **MANIFOLD** program for this new structure below.

```
 1 //pragma include "hugo6.ato.h"
 2
 3 #include "MBL.h"
 4
 5 #include "rdid.h"
 6
 7 #define NPART 5
 8
 9 #define IDLE terminated(void)
10
11 manner init_tb atomic.
12
13 manifold RD(port in filename) atomic {internal.}.
14
15 manifold PRENEX() atomic {internal.}.
16
17 manifold SPLIT()
18   port out qfprop.
19   port out qvarlist.
20   atomic {internal.}.
21
22 manifold CNFHH() atomic {internal.}.
23
24 manifold JOIN()
25   port in qvarlist.
26   port in ecnf1.
27   atomic {internal.}.
28
29 event divide, mission_accomplished, goon_uqe.
30
31 manifold UQE(port in npart)
```

```
  port out workers.
  atomic {internal. event divide, mission_accomplished, goon_uqe.}.

manifold WORKER(event ready) atomic {internal.}.

manifold HH() atomic {internal.}.

manifold PR() atomic {internal.}.

/************************************************/
manifold M_UQE()
{
  event wait, ready.

  process uqe is UQE(NPART).

  priority wait > ready.

  auto process i is variable.

  auto process count is variable(0).

  auto process worker is variable[NPART].

begin:
  for i = 0 while i < NPART step i = i + 1 do
    worker.input[i] = &WORKER(ready);
  (
    MES("begin"), activate(uqe), getunit(input) -> uqe, IDLE
  ).

divide:
  {
    save *.
    begin:
      MES("divide");
      for i = 0 while i < NPART step i = i + 1 do
      (
        getunit(uqe.workers) -> $worker.output[i],
        MES("job to worker", i)
      );
      post(wait).
  }.

wait:
  (MES("wait"), preemptall, IDLE).

ready.*:
  (MES("ready"), count = count + 1);
  EMES(count);
  if (count == NPART) then (
    count = 0, raise(goon_uqe),
    post(wait),
    MES("goon_uqe has been raised and wait has been posted")
  ) else (
    post(wait),
    MES("wait has been posted")
  ).

mission_accomplished:
  MES("mission_accomplished");
  for i = 0 while i < NPART step i = i + 1 do
    deactivate($worker.output[i]).

end:
  (MES("end"), output = uqe.output).

}
/************************************************/
manifold CNF()
{
  process split is SPLIT.
  process cnfhh is CNFHH.
  process join is JOIN.
  begin:
    (
      MES("begin"),
      activate(split, cnfhh, join),
      input -> split,
               split.qfprop -> cnfhh -> join.ecnf1,
               split.qvarlist -> join.qvarlist,
      join -> output
    ).

  end: MES("end").

}
/************************************************/
manifold Main(process arg)
{
  process rd is RD(tuplepick(arg, 2)).
  process prenex is PRENEX.
  process cnf is CNF.
  process m_uqe is M_UQE.
  process hh is HH.
```

```
128   process pr is PR.
129
130   begin:
131     (
132       MES("begin"),
133       init_tb,
134       activate(rd, prenex, cnf, m_uqe, hh, pr),
135       rd -> prenex -> cnf -> m_uqe -> hh -> pr
136     ).
137
138   end: MES("end").
139 }
```

The UQE manifold is slightly different than its previous version presented earlier. It now has a parameter (line 31) that indicates in how many pieces the list is be cut, it has a user-defined port named workers (line 32) and it can raise and receive events (line 33). On the other hand, the structure of the manifolds CNF (line 102) and Main (line 121) has not changed except that now M_UQE is used in the pipeline of Main (line 134). Below, we give separate descriptions of the new manifolds. These descriptions also explain how they cooperate with each other according a master/worker protocol.

The new UQE manifold is still implemented in C (line 31) but it now behaves as follows:

1. Read a term pointer from the input port.
2. Do some computational work, but when a list of terms must be simplified, don't do it yourself. Instead raise the event divide (this is done by an AP_Raise call in the C code of UQE) to signal a coordinator (this will be the manifold M_UQE) to delegate some work (we see later how this event will be handled by M_UQE).
3. Divide the list of terms in NPART pieces (which is set to 5 on line 7) and write the pointers to the sub-lists to the user-defined output port workers (line 32).
4. Wait until the delegated work is done. This will be noticed by receiving the goon_uqe event in the event memory, which is raised by the coordinator M_UQE (line 83).
5. Merge the sub-lists of terms into one big list.
6. Repeat steps 2, 3, 4, and 5 as many times as needed. When UQE is done with all its work, it raises the event mission_accomplished (by an AP_Raise call in the C code) to signal the coordinator M_UQE that it is done.

The WORKER manifold is implemented in C (line 35) and it behaves as follows:

1. Read a pointer to a sub-list of terms from the input port.
2. Perform the simplification on the sub-list.
3. Raise (by a call to AP_Raise in the C code) the event received as a parameter (line 35) when the simplification transformation on the sub-list is done.
4. Repeat steps 1, 2, and 3 until deactivated by M_UQE (line 94).

The M_UQE manifold is implemented in the MANIFOLD language (line 42) and behaves as follows.

In its begin state, NPART workers are created (line 58) whose references (denoted by &) are stored in an array named worker (line 54). Further, we activate in this state the already created uqe (respectively on lines 60 and 46) and take out a unit from the input port, representing a term pointer so that uqe can consume it (line 60). We remark here that the only reason why we store the process references in an array is to use it in a "for loop" and other syntactic constructs of the MANIFOLD language. With the dereference operator (denoted by $) we can always make a handle back to the process instance as shown on line 70.

Sooner or later, UQE raises the divide event (see step 2 in the description of the UQE manifold) which causes a state switch to the divide state (line 63). In that state, the pointers to the sub-lists are transported to the workers (line 70) so that they can do their simplification transformations

in parallel. When the workers are ready, they raise the `ready` event (see step 3 in the `WORKER` manifold). These events are counted in the `ready` state (line 79). When there are `NPART` ready events counted (line 82) all workers are done simplifying their sub-lists. In this way, we create a synchronization point in the application. After this rendez-vous of workers, the event `goon_uqe` is raised (line 83) to signal `UQE` to merge (in shared memory) the sub-lists into one big list (step 5 in the `UQE` manifold). Thereafter, the `wait` event is posted (line 84) which causes a state switch to that state (line 76). There, two things can happen: either another `divide` event is received, in which case the sequence of actions just described starts again, or the `mission_accomplished` event is received, in which case we deactivate the workers (line 94).

Note that in the `M_UQE` module (in principle this counts for every manager module) we only discuss coordination issues. In this sense, it forms an isolated piece of code that we can consider as the realization of a cooperation model (in this case a master/worker model). Therefore, it is irrelevant what it coordinates. Our `M_UQE` manifold can just as happily orchestrate the cooperation of any pair of processes that have the same input/output and event behavior as `UQE` and `WORKER` do, regardless of what computation they perform (see also [ABBE96] for this phenomena).

6. Some Experiments

In this section we describe a family of test assertions and show how we transform them into a family of closed quantified propositions. We also give the general format of these propositions, verify them, and give some performance results.

6.1 The Test Assertion

To verify the implementation of the proposition solver we use the following family of assertions which are parameterized by n.

> Each bit string of length n is a standard binary representation of an integer between 0 and $2^n - 1$.

The formal definition of this assertions is as follows:

$$\forall \vec{b} \in B_n.\ (\exists k \in N : 0 \leq k < 2^n).\ \vec{b} = \vec{r_n}(k)$$

where B_n is the set of possible bit strings of length n and $\vec{r_n} : N \mapsto B_n$ is the standard representation of the integers on a bit string of length n.
We can transform each of these assertions to a closed quantified proposition. For convenience we take $n = 2$. The predicate $0 \leq k < 2^2$ is equivalent to the logical formula

$$k = 0 \vee k = 1 \vee k = 2 \vee k = 3.$$

Let $[x_0 x_1]$ be a bit string of length 2. The transformation of this formula to the bit string format is now as follows:

$$[x_1 x_0] = [00] \vee [x_1 x_0] = [01] \vee [x_1 x_0] = [10] \vee [x_1 x_0] = [11].$$

If we work it out, this is equal to

$$x_1 = 0 \wedge x_0 = 0 \vee x_1 = 0 \wedge x_0 = 1 \vee x_1 = 1 \wedge x_0 = 0 \vee x_1 = 1 \wedge x_0 = 1.$$

The assertion (n=2) is that for all $[x_1\ x_0] \in \{$ [00], [01], [10], [11] $\}$ this formula must be equal to *true*. The last step is the transformation of this assertion to a closed quantified proposition. Let 1 represent *true* and 0 represent *false*, then

> the formula "$x_1 = 1$" becomes "$x_1 = \mathit{true}$" which is the same as the proposition "x_1", and
>
> the formula "$x_1 = 0$" becomes "$x_1 = \mathit{false}$" which is the same as the proposition "$\neg x_1$".

With similar results for x_0 instead of x_1, the assertion (n=2) becomes the following closed quantified proposition:

$$\forall x_1 . \forall x_0 .\ (\neg x_1 \wedge \neg x_0 \vee x_1 \wedge \neg x_0 \vee \neg x_1 \wedge x_0 \vee x_1 \wedge x_0).$$

6.2 The General Format of the Test Assertions
The general format of the test assertions $\phi(n)$ is as follows:

$$\forall \vec{x}.\ \bigvee_{i=0}^{2^n-1} \bigwedge_{j=0}^{n-1} \neg_j^i\ \vec{x}$$

where $\forall \vec{x}$ stands for the row

$$\forall x_{n-1} \ldots \forall x_0$$

and

$$\neg_j^i\ \vec{x} = \begin{array}{ll} x_j & \text{if } bit_j(i) = 1 \\ \neg x_j & \text{if } bit_j(i) = 0. \end{array}$$

$bit_j(i)$ returns the value of the jth bit of the binary representation of i on a bit string of length n. $\bigvee$ and $\bigwedge$ are, respectively, the generalized $\vee$ and $\wedge$ operators. It is evident that the result of the evaluation of $\phi(n)$ must be *true*.

6.3 Verification and Performance Results
We have verified the family of assertions for $n = 1$ up to 10. They all evaluate to "true". We did these experiments on SGI 16 processor machine of which only 5 processors were available for general use (that is why we set `NPART` to 5, line 7). The results of our performance measurements for the two cases (i.e., with and without the master/worker protocol) are summarized in table 1.

n	*approach I*	*approach II*
1	4.926	7.059
2	3.729	6.732
3	3.969	6.536
4	3.865	7.364
5	4.397	10.419
6	5.453	11.993
7	8.738	17.546
8	22.227	40.578
9	81.109	124.262
10	372.855	465.731

Table 1: Elapsed time (in seconds) for the 10 assertions for the two approaches (approaches I and II are, the pipeline model without and with the master/worker protocol respectively).

Note that the approach with the master/worker protocol does not improve the performance of the application. This is no surprise when we recall that we only added a hierarchic coordination layer to the application without the possibility to exploit its performance advantage through parallelism. Due to the coarse-grain lock for the thread-unsafe term library, in fact nothing is done in parallel. With a new thread-safe library we expect other results. For instance, we remark here that MANIFOLD has been successfully used to implement a parallel version of a semi-coarsened multi-grid Euler solver algorithm, using a similar master/worker protocol as the one described here. In that case, all programs were thread-safe, and the performance was improved from almost 9 to over 2 hours on a 4 processor machine [EK98].

7. Conclusions

Our experiment using MANIFOLD to coordinate the interprocess communication in a proposition solver indicates that this coordination language is well suited for this kind of tasks. The highly modular structure of both MANIFOLD programs in our case study and the ability to use the separately developed functional units is remarkable.

The unique property of MANIFOLD that enables such high degree of modularity is inherited from its underlying IWIM (*Idealized Worker Idealized Manager*) model in which communication is set up from the *outside*. The core relevant concept in the IWIM model of communication is isolation of computation responsibilities from communication and coordination concerns, into separate pure computation modules (as the functional units in the proposition solver written in C) and pure coordination modules (as `Main`, `CNF` and `M_UQE`).

The modularity of MANIFOLD also offers the possibility to introduce concurrency step by step. We can therefore proceed as follows. We initially plug a block of code as a monolithic computing process into a concurrent structure (as we did with `UQE`) to obtain a running parallel/distributed application. As more experience is gained through running the new application, computation bottlenecks may be identified (`UQE`). This may lead to replacing some such monolithic blocks of code with more MANIFOLD modules (`M_UQE`) that coordinate the activity of smaller blocks of computation code (`WORKER`), in a new concurrent sub-structure.

Another important advantage of MANIFOLD is that it makes no distinction (from the language point of view) between distributed and parallel environments: the same MANIFOLD code can run in both as we show in our "toy" application in section 2.

All these features make MANIFOLD a suitable framework for construction of modular software on parallel and/or distributed platforms.

Acknowledgments

We thank Farhad Arbab for his suggestions to improve this paper and Freek Burger for his programming advice.

References

[ABBE96] F. Arbab, C.L. Blom, F.J. Burger, and C.T.H. Everaars. Reusable coordinator modules for massively concurrent applications. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proceedings of Euro-Par '96*, volume 1123 of *Lecture Notes in Computer Science*, pages 664–677. Springer-Verlag, August 1996.

[Arb95] F. Arbab. Coordination of massively concurrent activities. Technical Report CS–R9565, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, November 1995. Available on-line http://www.cwi.nl/ftp/CWIreports/IS/CS-R9565.ps.Z.

[Arb96] F. Arbab. The IWIM model for coordination of concurrent activities. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 34–56. Springer-Verlag, April 1996.

[Arb97] F. Arbab. The influence of coordination on program structure. In *Proceedings of the* 30^{th} *Hawaii International Conference on System Sciences*. IEEE, January 1997.

[BK96] Jan Aldert Bergstra and Paul Klint. The ToolBus coordination architecture. In P. Ciancarani and C. Hankin, editors, *Coordination Language and Models*, number 1061 in LNCS. Springer Verlag, 1996.

[EK98] C.T.H. Everaars and B. Koren. Using coordination to parallelize sparse-grid methods for 3D CFD problems. *Parallel Computing*, 24(7):1081–1106, July 1998. special issue on Coordination languages for parallel programming.

[GC92] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communication of the ACM*, 35(2):97–107, February 1992.

[GKvV95] J.F. Groote, J.W.C. Koorn, and S.F.M van Vlijmen. The safety guaranteeing system at station hoorn-kersenboogerd (extended abstract). In *Proc. 10th Annual Conference on Computer Assurance (COMPASS '95)*, pages 57–68, 1995.

[Gro97] J.F. Groote. The propositional formula checker HeerHugo. Technical report, CWI, 1997. Unpublished note.

[LW98] B. Lisser and J.v. Wamel. Specification of components in a proposition solver. In J.v. Wamel J.F. Groote, B. Luttik, editor, *Proc. 3th International Conference on Formal Methods for Industrial Systems*, pages 271–298. CWI, 1998.

[NBF96] Bradford Nicols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Cebastopol, CA, 1996.

[NS93] A. Nerode and R. Shore. *Logic for Applications*. Texts and Monographs in Computer Science. Springer-Verlag-Hall, 1993.